1 Moduly – knižnice v Pythone

Python podporuje modulárne programovanie formou knižníc nazývaných moduly. Základy práce s modulmi popisuje výborne Tutoriál Pythonu v oddiele Modules. Napríklad modul `pluslib` definovaný v súbore `pluslib.py` môže poskytovať funkciu `plus(a,b)` a premennú `XXcode`. Tieto môžu používať iné aplikácie.

```
#pluslib.py

#an artificial Python library
#for addition

def plus(a,b):
    "sample function"
    return a+b

XXcode = 2.7
```

```
#application.py

import pluslib

print "2+3=", pluslib.plus(2,3)
print "ab'+'cd'=",
pluslib.plus('ab','cd')

print pluslib.XXcode
```

Aby mohol byť importovaný, musí byť modul na ceste, kde Python hľadá moduly (`sys.path`). Všetky funkcie a premenné definované v importovanom module sú prístupné cez bodkovú notáciu

modul.funkcia alebo *modul.premenná*.

Modul je tiež možné importovať pod novým menom (napríklad kratším)

```
import numpy as np
z = np.zeros(4)
```

Ak nehrozí kolízia názvov premenných, funkcií (alebo modulov definovaných pod modulom — viz. nápoveda k `__init__.py`), tak je možné importovať z modulu všetko

```
from pluslib import *
print plus(2,XXcode)
```

alebo iba vybrané položky

```
from pluslib import plus
print plus(92,200)
```

POZOR! Ak modul `M` už bol importovaný (v spustenom interprete) a vy ho potom zmeníte, tak volanie `import M` nový import nevykoná! Je treba zavolať

```
reload(M)
```

2 Matice v Pythone

Dvoj- a viacrozmerné polia je možné v Pythone reprezentovať vnorenými zoznamami

```
MatA = [[1,2,3], [4,5,6]]
MatB = [[7,8,9], [10,11,12]]
```

Ale operácie s takýmito štruktúrami sú v Pythone pomalé

```
MatA = [100*[100*[1]]
MatB = [100*[100*[2]]
MatC = [100*[100*[0]]
for i in range(100):
for j in range(100):
MatC[i][j] = MatA[i][j] + MatB[i][j]
```

Preto bolo vyvinutých niekoľko knižníc pre prácu s (viacrozmernými) poľami. Najrozšírenejšie sú `Numeric` a `numpy`. `numpy` je nástupcom `Numeric`, takže je doporučované používať `numpy`. Tieto knižnice implementujú polia typov `int`, `float`, `complex` a ďalších ako triedy, kde položky sú uložené v pamäti "za sebou". To umožňuje implementovať operácie nad takýmito poľami efektívnejšie (napr. v jazyku C).

```
>>> from numpy import *
>>> a = array([1,2,3,4,5])      #celociselny vektor zo zoznamu
>>> b = zeros(4)                #nulovy vektor dlzky 4 (realne cisla)
>>> c = ones(4)                 #vektor jedniciek dlzky 4 (realne cisla)
>>> D = array([[1,2], [3,4]])   #celociselna matica zo zoznamu
>>> f = zeros((3,4),int)        #nulova matica 3x4, celociselna
>>> ff = zeros((3,4),float)     #nulova matica 3x4, s realnymi cislami
>>> print ff
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
```

Výpis veľkej matice je skrátený

```
>>> e = ones((100,100))
>>> e
array([[ 1.,  1.,  1., ...,  1.,  1.,  1.],
       [ 1.,  1.,  1., ...,  1.,  1.,  1.],
       [ 1.,  1.,  1., ...,  1.,  1.,  1.],
       ...,
       [ 1.,  1.,  1., ...,  1.,  1.,  1.],
       [ 1.,  1.,  1., ...,  1.,  1.,  1.],
       [ 1.,  1.,  1., ...,  1.,  1.,  1.]])
```

2.1 Základné operácie s poľami

Základné operácie aplikované po prvkoch je možné robiť naraz na celé polia

```
>>> a = ones((3,2))
>>> b = array([[1,2], [3,4], [5,6]])
>>> a + b
array([[ 2.,  3.],
       [ 4.,  5.],
       [ 6.,  7.]])
>>> a-b
array([[ 0., -1.],
       [-2., -3.],
       [-4., -5.]])
>>> a * b  #!!! po prvkoch
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])
>>> a / b  #!!! po prvkoch
array([[ 1.        ,  0.5       ],
       [ 0.33333333,  0.25      ],
       [ 0.2       ,  0.16666667]])
>>> b[1] #riadok s indexom 1 (druhy)
array([3, 4])
>>> b[-2:] #posledne 2 riadky
array([[3, 4],
       [5, 6]])
>>> b[:,1] #stlpec cislo 1, ale v riadkovom vektore
array([2, 4, 6])
```

Pole má svoje rozmery v položke `shape` a svoj typ v položke `dtype`

```
>>> b.shape
(3, 2)
>>> b.dtype
dtype('int32')
```

2.2 Násobenie polí

Nech `a` a `b` sú dva vektory

```
>>> a = array([1,2,3])
>>> b = array([10,20,30])
```

`numpy` podporuje tri spôsoby ich násobenia:

1. Súčin po zložkách:

   $$c_i = a_i b_i, \quad \forall i$$

   v Pythone

```
>>> a*b
array([10, 40, 90])
```

2. Maticový súčin (*inner/dot product*):

$$c = \sum_i a_i b_i$$

   v Pythone

```
>>> dot(a,b)
140
```

3. Vonkajší súčin (*outer product*):

$$c_{i,j} = a_i b_j, \quad \forall i, j$$

   v Pyhtone

```
>>> outer(a,b)
array([[10, 20, 30],
       [20, 40, 60],
       [30, 60, 90]])
```

Súčin dvoch polí `a` a `b` v tvare `a*b` robí násobenie po zodpovedajúcich si zložkách, ak majú polia `a` a `b` rovnaký tvar.

```
>>> a = array([1,2,3])
>>> b = array([2,3,4])
>>> a*b
array([ 2,  6, 12])
```

Ak nemajú rovnaký tvar, ale sú pre `numpy` *kompatibilné*, tak sa polia upravia (kopírovaním, tzv. *broadcasting*) na spoločný tvar a zodpovedajúce si zložky sa vynásobia. Dve veľkosti dimenzií sú kompatibilné, ak sú zhodné alebo jedna z nich je 1. Pozor, riadkové pole má iba jednu dimenziu. Chýbajúce dimenzie sa zľava doplňujú jedničkami. Takže polia tvaru `(4,)` [1] a `(3,1)` sa upravia na spoločný tvar `(3,4)`:

Pole	Dimenzií	Tvar	Pole.shape
a	1	4	`(4,)`
b	2	3×1	`(3,1)`
a*b	2	3×4	`(3,4)`

[1] Rozmery `(4,)` je v Pyhtone n-tica s jediným prvkom. Bez čiarky za štvorkou by to bol iba výraz s hodnotou 4.

Prvé pole sa okopíruje trikrát v novej prvej dimenzii, aby sa dostalo na tvar 4×3, a v druhom poli sa okopíruje každá položka štyrikrát v druhej dimenzii.

```
>>> a = array([1,2,3,4])
>>> a
array([1, 2, 3, 4])
>>> a.shape
(4,)
>>> b = array([[10],[20],[30]])
>>> b
array([[10],
       [20],
       [30]])
>>> b.shape
(3, 1)
>>> a*b
array([[ 10,  20,  30,  40],
       [ 20,  40,  60,  80],
       [ 30,  60,  90, 120]])
```

Maticový súčin `dot(a,b)` :

1. pre jednorozmerné polia `a` a `b` počíta skalárny súčin;
2. pre dvojrozmerné polia `a` a `b` počíta súčin matíc;
3. pre viacrozmerné polia `a` a `b` počíta skalárne súčiny cez poslednú dimenziu poľa `a` a predposlednú dimenziu poľa `b`. Napríklad pre trojrozmerné polia

   `dot(a, b)[i,j,k,m] == sum(a[i,j,:] * b[k,:,m])`.

Vonkajší súčin `outer(a,b)` viacrozmerné polia vždy najprv "rozbalí" do jednorozmerných polí a až potom urobí vonkajší súčin.

3 Generovanie matíc

Okrem zadávania matíc ako štruktúrovaných zoznamov, generovania nulových a jedničkových matíc je možné vytvárať ďalšie špeciálne matice pomocou mnohých funkcií. Napr.

`eye(n)`	jednotková matica rádu `n`
`arange([start,] stop [, step])`	aritmetická postupnosť
`random.bytes(n)`	reťazec obsahujci `n` náhodných bytov
`random.normal(m=0.0, s=1.0, sz=None)`	náhodné čísla z normálneho rozdelenia m so strednou hodnotou `m` a smerodatnou odchýlkou `s`
`random.rand(d0,...,dn)`	náhodné čísla z uniformného rozdelenia z intervalu $\langle 0,1)$ v poli s rozmermi `d0`$\times\ldots\times$`dn`
`random.randn(d0,...,dn)`	náhodné čísla z normálneho rozdelenia so

	strednou hodnotou 0 a smerodatnou odchýlkou 1 v poli s rozmermi `d0`$\times \ldots \times$`dn`
`randint(low, high=None, size=None)`	náhodné celé čísla z intervalu `low`..`high`-1. Ak `high` je `None`, tak z intervalu `0`..`low`-1.
`permutation(n)`	náhodná permutácia čísel `1`,...,`n`
`permutation([a0,...,an])`	náhodná permutácia zoznamu `[a0,...,an]` – kópia pôvodného zoznamu
`shuffle([a0,...,an])`	náhodná permutácia zoznamu `[a0,...,an]` "na mieste"

Naviac pomocou funkcie `seed(i)` je možné inicializovať generátor náhodných čísel a tým zaručiť reprodukovateľnosť generovania pseudo-náhodných čísel. Tvar poľa sa dá ľahko meniť funkciou `reshape()`:

```
>>> arange(24)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23])
>>> arange(24).reshape(2,3,4)
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```